DEPENDENCY ANALYSIS OF FOR-LOOP STRUCTURES FOR AUTOMATIC PARALLELIZATION OF C CODE

Tim Jacobson
Mathematics and Computer Science Department
South Dakota School of Mines and Technology
Timothy.Jacobson@gold.sdsmt.edu

Gregg Stubbendieck
Mathematics and Computer Science Department
South Dakota School of Mines and Technology
Gregg.Stubbendieck@sdsmt.edu

Abstract

Dependency analysis techniques used for parllelizing compilers can be used to produce coarse grained code for distributed memory systems such as a cluster of workstations. Nested for-loops provide opportunities for this coarse grained parallelization. This paper describes current dependency analysis tests that can be used to identify ways for transforming sequential C code into parallel C code. Methods for searching nested for-loops and array references will be discussed as well as differences of each dependency test.

1 Introduction

Dependency analysis has predominately resided in the realm of compilers. The goal was to speed performance of processes by eliminating conflicts caused by data accesses. These dependency analysis techniques gained popularity in parallel compilers for shared memory systems. This paper is part of an on going project attempting to use existing dependency analysis techniques on a distributed memory system. The purpose will be to convert sequential C programs into parallel C programs that can be run on a cluster of workstations using a message passing protocol. One of the tenets of this project is that the parallel code be coarse grained as opposed to fine grained parallel code used with shared memory systems. One way that coarse grained parallelization can be achieved is by performing transformations of loop structures. Assuming the data dependencies found within the loop structures are intentional, dependency analysis will be used to catalog and verify that transformations do not change the algorithm. In order to perform dependency analysis, information must first be gathered about the loop structures. Once the loop structure information has been accumulated, it will be used to determine which dependency tests to perform. Each dependency test is designed for a specific type of data reference found in the loops. Pairs of data references will be compared together to determine if a dependency exists between them. Some dependency tests are known as exact tests because they can accurately determine dependence or independence. Due to the complexity of some data references, tests performed on them must use symbolic representations of dependencies. The simplification of complex data references may only allow dependency tests to approximate dependencies. The following paragraphs will describe specific dependency analysis techniques used on for loops of C code and the difficulties that are encountered.

2 Classifying Data Dependencies

A data dependence is any statement or set of statements that access the same memory location where at least one of the accesses is writing to that location. There are four kinds of dependence that can be produced [1] [2][3][4]. To explain each start with the following assumption: let s_1 and s_2 be two statements that access the same memory location. Statement s_1 occurs before s_2 in the program. The order in which the access occurs will define specific dependencies. Writing and reading from the two statements can be performed in four ways.

1. *True dependence*: In true dependence, sometimes called flow dependence, s_1 writes to the memory location while s_2 reads from the memory location.
2. *Anti dependence*: In an anti dependence, s_1 reads from the memory location while s_2 writes to that location.

3. *Output dependence*: In an output dependence, both s_1 and s_2 write to the same memory location and it assumed there is no other access to that memory location in the time between s_1 and s_2.
4. *Input dependence*: In an input dependence, both s_1 and s_2 read from the same memory location. Input dependence does not fit the definition of a data dependence because there is no writing involved. However, some authors feel the need to mention it for completeness.

It should be noted that statements s_1 and s_2 may be the same statement. Classifying the dependence between two statements is important for analysis, since different dependencies will be parallelized in different ways.

In order to classify the dependencies between two statements, an algorithm must be designed that will parse a set of instructions. Optimizing software tends to focus attention on loop structures in program code. Several items of information relating to the loop structures found in a particular set of instructions will aid in determining dependencies of statements. Among these is information about the stride of each loop, the number of loops contained in a nested loop structure, and the level at which each statement resides within the loop nest.

When looking at loop structures it is valuable to know the stride the loop performs for the iteration. The stride value is referred to as the *iteration number* [3]. Knowing if the iteration strides in an increasing or decreasing direction will also be of value. In figure 1, loop i has an iteration number of 1 and has a positive stride. Loop j has an iteration number of 2 with a positive stride. Loop k has an iteration number of 1 with a negative stride.

Figure 1: Nested for-loops with different strides

```
for(i = 0; i < 10; i++)
   for(j = 0; j < 10; j+=2)
      for(k = 10; k > 0; k--)
         A[0][i][j] = A[10][i][k];
```

When analyzing a nested loop structure, dependency tests need a way of keeping track of what level the statements are nested in called the *nesting level* [3]. Finding the nesting level is a simple counting. The outer most level is considered level one and each level inside is counted incrementally. Each statement will be cataloged as to which level it lies in the nest. Once statements are identified in the nest, the focus will turn to how those statements change from iteration to iteration.

Each statement lying in a nested loop structure contains an *iteration vector* that defines the iteration values of all the loops at that given point [3]. For a nest of n loops, the *iteration vector* $\alpha = \{a_1, a_2, a_3, .. a_n\}$, where $a_k : 1 \leq k \leq n$, represents the iteration number for the loop at nesting level k. The set of all iteration vectors for a given

statement is called the *iteration space* of the statement. The iteration space gives a complete view of how memory is being accessed by a statement throughout the lifetime of the nested loop. Referring to figure 1, the statements have the following iteration vectors:

A[0][i][j] = {{0,0,0}, {0,0,2}, {0,0,4}, {0,0,8}, {0,1,0}, {0,1,2}, … ,{0,9,8}}

A[10][i][k] = {{10,0,10}, {10,0,9}, {10,0,8}, {10,0,7}, {10,0,6}, … ,{10,0,1}}

Notice that there is no dependency between the two statements, since the first always contains the constant 0 in the first array reference and the second always contains the value 10. There is no possibility for the two statements to refer to the same memory location.

Now that information has been gathered about each statement, the next step is to begin identifying dependencies between statements. In a nest of loops there is a dependence from statement s_1 to s_2 such that $\alpha = \beta$ where α is a particular iteration vector for s_1 and β is a particular iteration vector for s_2 [3]. Thus statement s_1 accesses a memory location on iteration vector α and s_2 accesses the same memory location on iteration vector β. It would be possible perform comparisons between each iteration vector for two statements, but may take a very long time for loops of any significant size. In fact, if the loop structures are of a size that would be worth parallelizing, then performing a one to one comparison for each iteration vector will be ridiculously long.

One way to lessen the computation is to view iteration vectors in a more abstract way. When a loop dependence occurs between iteration vectors α and β, then the distance between them is defined as the *distance vector* $d(\alpha, \beta)$, which is the difference between the two vectors [3].

$$d(a, b)_k = b_k - a_k .$$

After obtaining the distance vector, the values can be further abstracted to a set of symbols representing the three directions. The *direction vector* $D(\alpha, \beta)$ is defined as:

$$D(\alpha, \beta) = \begin{cases} \text{“<” if } d(a, b)_k > 0 \\ \text{“=” if } d(a, b)_k = 0 \\ \text{“>” if } d(a, b)_k < 0 \end{cases}$$

These symbols will be used to verify that transformations performed allow dependencies to be maintained. For example:

```
A[i + 1][j][k - 1] = A[i][j][k];
```

The distance and direction vectors will be (1,0,1) or (<,=,>). The type of dependence should still exist after the transformation to parallel code otherwise the algorithm has changed and may produce incorrect results.

As mentioned earlier, the dependence between two statements occurs when both of them access the same memory location and at least one of them writes to that location. When considering loop structures, the dependency between two statements may occur in the same iteration or separate iterations of a loop. If the dependency is found in the same iteration of the loop it is called a *loop-independent dependence* [3]. For example: if s_1 and s_2 are two statements in the same level of a loop and both access memory location M during that iteration of the loop. Then on the next iteration, both statements will access a new memory location N. Since the iteration does not access any memory locations from previous iterations, the iteration is independent of all others. A *loop-carried dependence* [3] occurs when a statement accesses a memory location and then on another iteration there is a second access of that memory location. Thus, s_1 will access memory location M and then on a subsequent iteration s_2 will also access memory location M. Again, s_1 and s_2 may be the same statement where one is a write and one is a read.

3 Simple dependency tests

In describing variables in nested loop structures, there are two common terms *index* and *subscript* [2][3]. The index is a variable that is used in loop structures to iterate the loop. Indices are commonly used in array references and will be the focus of many of the dependency analysis algorithms. When an index is used in array reference, it is called a subscript. For example, a two dimensional array A[i][j], there are two subscripts. Assuming the array reference is nested in a double loop with index variables i and j. Much of the difficulty in dependency analysis is determining if the subscripts of two array references match.

To expand on the focus of indices and subscripts will be the question of how many indices are found in the subscripts of an array reference? The number of indices will be called the *complexity* of the array [2][3]. There are three levels of complexity that are commonly used for dependency analysis. The first is the *zero index variable* (ZIV). A ZIV is an array reference, which does not use index variables in a particular subscript pair. The second is the *single index variable* (SIV). The SIV uses one index variable in a particular subscript pair of an array references. Finally, the third is called the *multiple index variable* (MIV). The MIV occurs when more than one index variable is reference in a particular subscript. Note that any pair of multi-dimensional array references may contain any one or several complexity values in the subscripts. For example, the nested loop mentioned above in figure 1 contains a ZIV in the first subscript, a SIV in the second subscript and a MIV in the third subscript. There are different dependency tests that are performed for ZIV, SIV, and MIV subscripts.

In addition to knowing the complexity of subscripts, the *separability* of the subscripts will need to be known [2][3]. A subscript pair is separable if all the indices used in it are

exclusive to that subscript. If any two different subscript pairs share an index variable, then those subscripts are considered *coupled*. For example:

```
A[i][j][j] = A[i][j][k];
```

The first subscript is separable because it only contains the index i, but the second and third are coupled because they share the variable j. Before dependency testing can begin, a pair of array references must be broken into separable and minimally coupled subscripts. For the separable subscripts, there are methods of getting exact answers from dependency tests. Due to complexity, coupled subscripts have more difficulty in finding exact answers. In the event that an exact answer cannot be determined, a more conservative ruling will be made. There is the potential that a dependency may be detected that does not really exist. However, the opposite is not true. Dependency tests do not declare independence between subscripts when one really exists.

3.1 **Testing for ZIV**

One of the simplest dependency analysis tests is to compare a pair of subscripts that contain no index variables. Referring to figure 1, subscripts 0 and 10 will be tested for ZIV. Since these subscripts hold integer values they can easily be compared by subtraction. If the difference is non-zero there is no dependency. If the subscripts held variables, then either the value of the variables are known at runtime or they are unknown. If the variables are known, the test can compare the values similarly by taking the difference. If they are unknown, then there are two possible situations. One is that the same variable is used in both subscripts and may have an arithmetic operator associated with it. A symbolic comparison of the variables can be made to verify they differ by an arithmetic operator. Otherwise one or both variables are initialized at runtime and a conservative approach is made to assume they are the same value, because it cannot be proven they are different.

3.2 Testing for SIV

SIV tests may be divided into several sub-categories based on information associated with the common index of a subscript pair. An index may be combined with a number or variable through an arithmetic operator in a subscript. SIV subscripts can be viewed as linear expressions of the form: $ai + c$, where a and c are either numbers or variables representing numbers and i is the index. A pair of subscripts can be written as two linear functions: $f(i) = a_1 i + c_1$ and $g(i) = a_2 i + c_2$. Because these are linear, they can be plotted as lines in a rectangular coordinate system.

The two major classifications of SIV subscript pairs will be parallel lines, or *strong SIV subscripts*, and non-parallel lines, or *weak SIV subscripts*[2][3]. Weak SIV testing may be further divided into sub-categories that will be mentioned later. Note that the lines displayed graphically represent more than the actual data references. The only locations of concern are points on the lines that actually have integer coordinates since subscripts

must be integer values. Dependencies occur between the two lines only when there exists a horizontal line that can intersect both subscript lines and find integer coordinates at those intersections. The *horizontal line test* is the equivalent of finding two references to the same memory location.

3.2.1 Strong SIV test

The strong SIV test is for comparing two subscripts that contain an index i of the form $a_1 i + c_1$ and $a_2 i + c_2$. Since these can be represented as parallel lines, they have the same slope or $a_1 = a_2$. The dependence distance between the two lines is merely the difference between any two points intersected by a horizontal line. The horizontal difference can be calculated by the finding the intercepts of the horizontal axis.

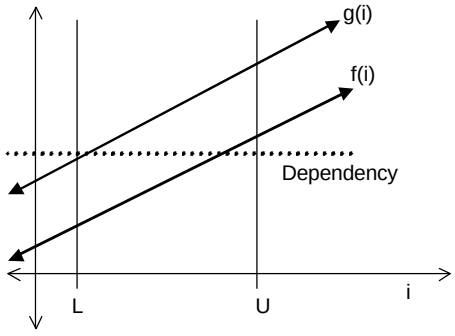
Using the equations $f(i) = a_1 i + c_1$ and $g(i) = a_1 i + c_2$,

the horizontal intercepts are $-\frac{c_1}{a_1}$ and $-\frac{c_2}{a_1}$.

The difference between them is $d = \frac{c_1 - c_2}{a_1}$,

where d is the dependence distance[3]. A dependence exists only if d is an integer and |d| < U – L , where U and L are the upper and lower bounds of the index I, see diagram 1. If the dependency exists then it will be classified as <, >, or = by the value of d compared to zero.

Diagram 1: Strong SIV test



3.2.2 Weak SIV tests

The weak SIV test occurs when the two subscripts in question have values that form intersecting lines. For subscripts $a_1 i + c_1$ and $a_2 i + c_2$, $a_1 \neq a_2$ or the slopes of the lines are not the same [3]. There are a couple special cases of weak SIVs that are easy to test

for dependencies. These include the scenario where one of the subscripts forms a horizontal line called a *weak-zero SIV subscript*. The other special case is when the subscripts form lines that have negative slopes of each other called the *weak crossing SIV subscripts*. Both of these cases have fast algorithms for determining dependence. If the subscripts don't fall into one of these categories, they must undergo the more expensive *exact SIV test*.

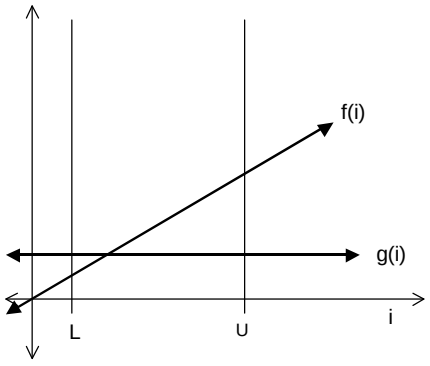
3.2.3 Weak-zero SIV test

The weak-zero SIV test is for comparing two subscripts where only one contains an index variable. The general form of the subscripts is $a_1 i + c_1$ and $a_2 i + c_2$, where either $a_1 = 0$ or $a_2 = 0$. The expressions can form two linear functions where one is a horizontal line and the other is non-horizontal. If the intersection of the two lines has integer coordinate for i and that location is within the lower and upper bounds of the index, there is a dependency, see diagram 2. Assuming that $a_2 = 0$, dependency can be calculated by the expression

$$i = \frac{c_1 - c_2}{a_1} \qquad [3].$$

Weak-zero dependencies have great potential for parallelization because only one location actually has a dependency and which can treated as a special case outside the loop.

Diagram 2: Weak-zero SIV test

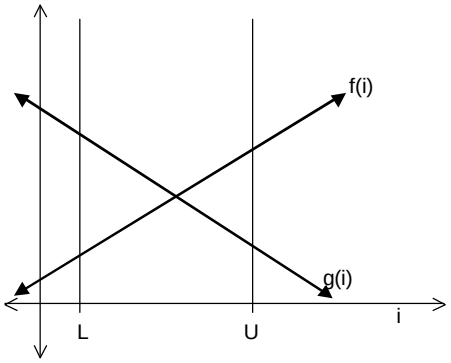


3.2.4 Weak-crossing SIV test

The weak-crossing SIV test is the situation where the subscript pair contains a common index but each is the negative of the other. The general form is the expressions $a_1 i + c_1$ and $a_2 i + c_2$ with $a_1 = -a_2$ [3]. Represented as functions produces two lines that are mirror images of each other reflected about a vertical line passing through the intersection of the lines. The point at which they intersect is the location of interest. If

the intersection point has integer coordinates or the horizontal coordinate is half way between two integer values, then there are dependencies to identify. As before, the dependencies must occur between the lower and upper limit to be valid, see diagram 3. Two dependencies may be found. One will be a flow dependency and the other anti-dependency with a change from one to the other at the point of intersection.

Diagram 3: Weak-crossing SIV test



3.2.5 Exact SIV testing

When none of the previously mentioned techniques apply to SIV subscripts, it is necessary to implement the exact SIV test. The test is possible by finding all the solutions to linear diophantine equations. A linear diophantine equation in two variables is one of the form ax + by = c, where a, b, and c are integers.

Given the subscript pair: $a_1 i + c_1$ and $a_2 i + c_2$

Re-label as: $ax + c$ and $by + d$

Now the subscripts will be set equal to each other. $ax + c = by + d$

Changed the order to produce a diophantine equation. $ax - by = d - c$

Find the greatest common denominator between a and b, $\gcd(a,b) = g$.

Divide the equation by *g*. $\frac{ax}{g} - \frac{by}{g} = \frac{d-c}{g}$

Since *g* divides *a* and *g* divides *b*, then *a/g* and *b/g* are integers. If (*d*-*c*)/*g* is also an integer, then there are integer solutions to x and y which will indicate dependencies. Finding those dependencies requires knowing the linear combination:

$$an_a + bn_b = g\ .$$

The greatest common denominator and linear combination can be calculated using the Euclidean Algorithm. Finally, the following equations can be derived:

$$x_k = n_a\left(\frac{d-c}{g}\right)+k\frac{c}{g} \qquad \text{and} \qquad y_k = n_b\left(\frac{d-c}{g}\right)-k\frac{d}{g} \qquad [3].$$

The variables x_k and y_k are specific solutions of the index i where there is a dependency between the subscripts. The variable k is any integer value and guarantees the dependence will occur for the subscript pair at that value. There are an infinite number of values of k that can be used, but the only ones of concern will be those that provide solutions x_k and y_k within the boundaries of the loop. It is possible that dependencies will only exist outside the boundaries. Testing for values within the lower and upper boundaries will tell if a dependency truly exists. The exact SIV test requires much more work to be certain of dependencies, but has the advantage of identifying all dependencies for linear SIV subscripts.

3.3 Testing for MIV

A MIV is any subscript pair that contains more than one index. For example:

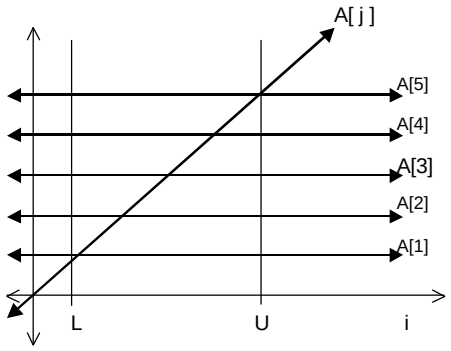
Figure 2: MIV in a single subscript

```
for(i = 0; i < 5; i++)
   for(j = 0; j < 5; j++)
      A[i] = A[j];
```

There are multiple occurrences of dependencies that take place on different iterations of the outer loop, see diagram 4. However, it is not as obvious to derive the linear equations for A[i] using techniques mentioned previously. Use of Diophantine equations also becomes difficult or computational expensive with multiple variables. One of the desires of dependency analysis is to avoid having to test every possible scenario between the subscripts in question. There are several dependency tests that have been designed specifically for MIV subscripts. This paper will cover some of the basic concepts.

The subscripts described in the figure 2 are known as Restricted Double Index Variables (RDIV). The general form for RDIV subscripts is $a_1 i + c_1$ and $a_2 j + c_2$ [2][3]. Since both subscripts are linear expressions, testing for dependencies in the RDIV is similar to the tests described for SIV. The RDIV test utilizes information about loop boundaries to augment the SIV tests.

Diagram 4: RDIV dependencies on different iterations



3.3.1 The Banerjee GCD test

The *greatest common denominator test* (GCD) is based on forming affine functions out of multiple index subscripts.

Let $A[f(x_1, x_2, x_3, ...x_n)]$ and $A[g(y_1, y_2, y_3, ...y_n)]$ be MIV subscripts.

Then $f(x_1, x_2, ..., x_n) = a_0 + a_1 x_1 + ... + a_n x_n$

and $g(y_1, y_2, ..., y_n) = b_0 + b_1 y_1 + ... + b_n y_n$.

These functions can be combined to form a Diophantine equation

$$a_1 x_1 - b_1 y_1 + ... + a_n x_n - b_n y_n = a_0 - b_0 \qquad [3].$$

If the $\gcd(a_1, ..., a_n, b_1, ..., b_n)$ can divide $a_0 - b_0$ then there is a dependency somewhere. However, the dependency may not be in the bounds of the array, so further testing will be needed. Also, it is not uncommon for the gcd of several numbers to be one, which divides everything. Thus the GCD test alone is not always exact. One way to improve the GCD test is to use Banerjee Inequalities. This technique abstracts the corresponding coefficients of the affine functions to inequality symbols described for dependence directions. These dependence directions can be formed into direction vectors and will be compared to $a_0 - b_0$ to determine in which loop the dependence occurs. The details of Banerjee GCD method are lengthy and will not be explained in this paper. The Banerjee GCD test is one of the oldest tests and is unofficially used as a benchmark by which many other MIV tests are compared.

3.3.2 The Delta test

One MIV test that is compared to the Banerjee GCD test is called the Delta test [2][3]. The Delta test is able to take constraints of SIV subscripts and incorporate them into MIV

subscripts with the hope of eliminating a common index. The MIV subscript may be reduced to a SIV or at the very least a less complex MIV. For example:

Figure 3: MIV in coupled subscripts

```
for(i = 0; i < 5; i++)
   for(j = 0; j < 5; j++)
      A[i + 1][i + j] = A[i][i + j];
```

The first subscript pair contains only the index i. Analyzing the first pair by the strong SIV test produces a dependence distance of 1. Now the Delta test will take the first subscript and subtract it from the second for both array references.

On the left hand side `(i + j) - (i + 1) = (j - 1)`.

On the right hand side `(i + j) - (i) = (j)`.

Thus the second subscripts have been reduced to `(j – 1) and (j)`.

The second subscripts can now be analyzed with the strong SIV test producing a dependence distance of –1. The distance vector of the two will be d(1,-1). The method of incorporating a subscript into another subscript is called *propagating constraints*. If the index that is propagated is completely eliminated from the subscripts, the Delta test will produce exact results. If the index cannot be completely eliminated, then at least the complexity has been reduced which will aid other MIV tests.

3.3.3 Other MIV tests

There are many other MIV tests that have been developed. Each has unique advantages and disadvantages. Some use the divide and conquer approach where subscripts are divided into separable and minimally coupled groups. However, analyzing subscripts individually and combining may produce false dependencies between subscripts. Other techniques try to overcome false dependencies by analyzing all the subscript pairs together, which adds computational complexity and expense. In an attempt to ease calculations and performance time, some tests abstract the elements of each subscript. Abstraction may also lead to less accurate results. Tests that are not exact are classified as *approximate tests*. Some of the MIV tests that are worth mentioning are the Fourier-Motzkin Elimination test, the Constraint-Matrix test, the λ-test, the Power test, the I-test and finally the Omega test [2][3][5][6]. These tests will be looked at in the future as part of the on going project.

4 Conclusion

This paper has described what data dependencies are and how they are detected in for-loops. In order for statements to be analyzed, information about the loops they are nested

in must first be gathered. This information is used to determine the types of data references contained in the statements. Often data references are arrays and the subscripts found in the arrays are the focus of most dependency analysis tests. Some of the dependency analysis tests are able to exactly determine if a dependency exists. Other tests use abstraction as a way of simplifying the analysis and may only produce approximate answers. The purpose of studying these dependency analysis techniques is to incorporate them into an automatic parallelization tool that will transform sequential C code into parallel C code that can be executed on a cluster of workstations. This parallelization will need to be coarse grained, but must maintain the dependencies that are inherent to the sequential version. Dependency analysis is the key to finding this coarse grained parallelization.

5 References

1. Kulkarni, D., & Stumm, M. (1993, June). Loop and Data Transformations: A Tutorial, Computer Systems Research Institute, University of Toronto, Toronto, Canada.

2. Goff, G., Kennedy, K., & Tseng, C. (1991, June). Practical Dependence Testing, Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, Toronto, Canada.

3. Allen, R., & Kennedy, K. (2002). Optimizing Compilers for Modern Architectures, San Francisco: Morgan Kaufmann Publishers.

4. Morgan, R. (1998). Building an Optimizing Compiler. Boston: Butterworth-Heinemann Publishers.

5. Pugh, W. (1992, August). A Practical Algorithm for Exact Array Dependence Analysis, Communications of the ACM, vol. 35, No. 8.

6. Muchnick, S. (1997). Advanced Compiler Design and Implementation. San Diego: Morgan Kaufmann Publishers.